A leveraged investment strategy using Deep Reinforcement Learning

Master's Thesis submitted

to

Prof. Dr. Wolfgang Karl Härdle

Humboldt-Universität zu Berlin

School of Business and Economics

Institute for Statistics and Econometrics

Chair of Statistics

by

Ilyas Agakishiev

(526854)

in partial fulfillment of the requirements

for the degree of

Master of Science

Berlin, April 21, 2019

Acknowledgement

Hereby, I would like to thank Prof. Dr. Wolfgang Karl Härdle for his supervision, Bruno Spilak for his help during my research and Z. Jiang et al. for providing the basis for the python code.

Abstract

This thesis describes a Deep Reinforcement Learning algorithm for portfolio management with multiple innovations. Its Reward function allows the investor maximize returns while regulating her risk preferences in terms of Maximum Drawdown. The risk is regulated by changes in the portfolio structure and by increasing or decreasing leverage.

The algorithm consists of two Convolutional Neural Networks (CNNs) - one is responsible for the weight distribution, the other for the leverage, ranging from 0 to 2.

Experiments have shown that, in bullish markets, the algorithm restricts itself and mostly complies with the defined target drawdown. In bearish markets, on the other hand, the algorithm leaves the market entirely. This is a major improvement compared to the algorithm by Jiang et al. (2017), which, having a different method of restricting trading, sometimes fails to do so in the same situations.

Contents

1 Introduction

Portfolio management is one of the most important tasks in quantitative finance. And, since the portfolio selection paper by Markowitz (1952), there have been rapid advancements in this field.

Many current portfolio management strategies require some assumptions, or a "model of the world". However, financial markets are very complex, and no model will account for everything.

A part of the solution is the application of Machine Learning, especially Deep Learning, as it can create a model of any complexity. Deep Learning in Finance is still young, but, over the last years, has seen some remarkable progress. Some of the most interesting applications of Deep Learning in this field are the analysis of Limit Order Books (Sirignano, 2016) and a Neural Network based stochastic volatility model (Luo et al., 2017), which can be used for forecasting.

Another achievement in Machine Learning is Reinforcement Learning. Since its discovery, one of the most common applications of it was teaching an algorithm, how to play games, such as backgammon, checkers, go (Silver et al., 2016) or video games (Farebrother et al., 2018). Portfolio management can also be considered as a game, so its usage makes sense. This area is in active research now. Papers by Jiang et al. (2017) and Liang et al. (2018) are two examples of how Reinforcement Learning can be used in Portfolio management.For most investors, a trade-off between return and risk is desired. The objective of the paper is to study, how Reinforcement Learning can be applied for this purpose. Just like in the Markowitz portfolio, the algorithm will try to maximize returns and, at the same time, hold the risk metric at a fixed level. The only difference is that we will use Maximum Drawdown as the risk metric instead of volatility, since many investors consider the former to be more important. A similar approach, with the Sharpe ratio as the risk metric, was attempted by Necchi (2016). In addition to that, a new approach is introduced that incorporated leveraging as an option to reduce risk or increase returns.

The structure of the thesis is as follows. Section 2 describes the basic methods used in the thesis, such as the concept of Convolutional Neural Networks and of Reinforcement Learning. Section 3 goes into the specifics of the model, describes the problem in quantitative terms,

shows the specific network structure and the way, Reinforcement Learning is applied. Section 4 shows the results of experiments and describes them. Finally, Section 5 comments further on the methods and the results and suggests topics for further research.

2 Methodology

2.1 Convolutional Neural Networks

2.1.1 Overview

Convolutional Neural Networks (CNNs) are a specific type of neural network that is mostly used to process grid-like data. Typical examples are audio, picture and video processing, i.e. the data can be temporal, spatial or both (Goodfellow et al., 2016). The name is given based on the mathematical operation "convolution", which is defined as the integral of the product of the two functions after one is reversed and shifted:

$$s(t) = (f * g)(t) = \int_{-\infty}^{\infty} f(a)g(t-a)da$$

Obviously, CNNs do not operate with integrals. Instead, they operate with so-called discrete convolution:

$$s(t) = (f * g)(t) = \sum_{a=-\infty}^{\infty} f(a)g(t-a)$$

This is a one-dimensional convolution operation. However, it is more likely that a convolutional layer is two-dimensional, and, to simplify calculations and avoid kernel flipping, cross-correlation is used instead of convolution:

$$S(t) = (K * I)(i,j) = \sum_{m}\sum_{n} I(i+m, j+n)K(m,n)$$

2.1.2 Structure and forward propagation

A rough structure of a convolutional layer looks like this:

- Convolution - the main operation, illustrated in Figure 2. Note that both the input and the kernel can have any possible size. The values of the kernel are "weights", initialized randomly and then trained. In addition to that, a kernel can have multiple filters - essentially multiple kernels with different weights for better feature extraction and multiple layered outputs.

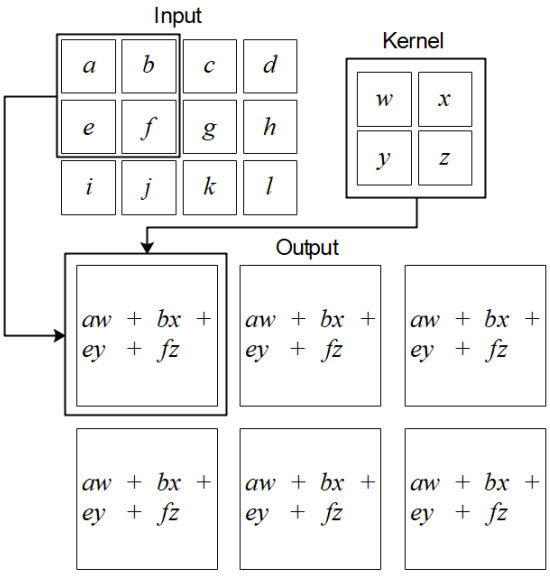


Figure 1: Illustration of a discrete convolution operation in practice

- Activation function - manipulates the output to ensure its non-linearity. Typical examples are the sigmoid function, hyperbolic tangent (tanh) and rectified linear unit (ReLU). In practice, ReLU almost always gives the best performance in the training process and is therefore used in this Thesis. It is defined as:

$$f(x) = x^+ = max(0, x)$$

- Pooling - modifies the output further by including values from the neighbouring input cells in the output cell. The most popular types of pooling are max pooling (includes the maximum value within a rectangular neighbourhood) and average pooling (calculates the average value within a rectangular neighbourhood). Max pooling is usually preferred, since it captures outlier information better. While pooling is important in such tasks as picture processing, it has no use in investing, and will not be included in this Thesis.

As mentioned earlier, Convolutional neural networks are used for effective processing of spatial and temporal data. Also, the structure of CNNs allows it to have a significantly smaller number of features, compared to a fully connected, dense network. That means that CNNs require less memory and less training time compared to other architectures. The downside is that CNNs have limited application cases, since it requires neighbouring values to be correlated in some way.

2.1.3 Backpropagation

Backpropagation is a method used to calculate a gradient that is needed in the calculation of the weights to be used in the network (Goodfellow et al., 2016). In other words, it is essential to train the network to capture the pattern in problems. In any network, a backpropagation step is calculated using the chain rule as shown below.

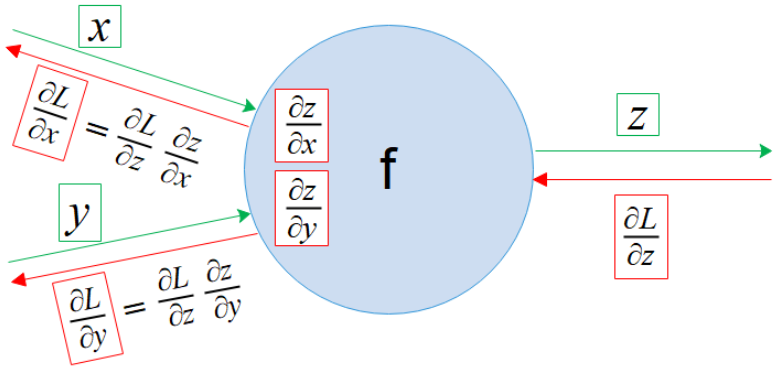


Figure 2: Illustration of a backpropagation step. The green arrow symbolize the forward propagation, while the red arrows show the backpropagation. The formulas in the red boxes are the gradients.

To show how backpropagation works for a Convolutional layer, assume a 3x3 input matrix X, a 2x2 filter (or kernel) W and a 2x2 output matrix H. Then the output matrix elements are calculated as:

$$h_{11} = W_{11}X_{11} + W_{12}X_{12} + W_{21}X_{21} + W_{22}X_{22}$$

$$h_{12} = W_{11}X_{12} + W_{12}X_{13} + W_{21}X_{22} + W_{22}X_{23}$$

$$h_{21} = W_{11}X_{21} + W_{12}X_{22} + W_{21}X_{31} + W_{22}X_{32}$$

$$h_{22} = W_{11}X_{22} + W_{12}X_{23} + W_{21}X_{32} + W_{22}X_{33}$$

In a backpropagation step, $\partial h_{ij} = \frac{\partial L}{\partial h_{ij}}$ is given, while $\partial w_{ij} = \frac{\partial L}{\partial w_{ij}}$ and ∂x need to be calculated. For ∂h_{ij}, the formulas are:

$$\partial W_{11} = X_{11}\partial h_{11} + X_{12}\partial h_{12} + X_{21}\partial h_{21} + X_{22}\partial h_{22}$$

$$\partial W_{12} = X_{12}\partial h_{11} + X_{13}\partial h_{12} + X_{22}\partial h_{21} + X_{23}\partial h_{22}$$

$$\partial W_{21} = X_{21}\partial h_{11} + X_{22}\partial h_{12} + X_{31}\partial h_{21} + X_{32}\partial h_{22}$$

$$\partial W_{22} = X_{22}\partial h_{11} + X_{23}\partial h_{12} + X_{32}\partial h_{21} + X_{33}\partial h_{22}$$

The backpropagation for ∂x works in a similar fashion.

The backpropagation of the ReLU function, given its definition, is easy to compute:

$$\frac{d}{dx}ReLU(x) = \begin{cases} 0, & \text{if } x < 0, \\ 1, & \text{otherwise.} \end{cases}$$

2.1.4 Regularization

With every complex algorithm, there is always the issue of overfitting - the network learns the patterns of the training set so well, such that they do not generalize for any new data. For that reason, regularization is applied - a set of measures to prevent overfitting from happening. The three most popular regularization methods are:

- Early stopping - the easiest way to prevent overfitting is forcing the network to stop training at a certain point, i.e. after a certain number of iterations. One method of determining the number is by checking the performance, measured by the loss value, on the validation set - at first, it will improve, but then, depending on the model, it will stop improving or start to get worse, the higher the number of iterations is (Prechelt, 2012).

- L2 (Tikhonov) regularization - modifies the cost function such that excessive weights are actively punished. The resulting formula is:

  $$J_{reg} = J + \frac{\lambda}{2m}\sum_i w_i^2,$$

  where m is the number of samples in batch and *lambda* is the regularization parameter. The concept was first introduced by Tikhonov (1943) to regularize ill-posed problems. When applied to linear regressions, the regression is called Ridge Regression (Hoerl and Kennard, 1970).

- Dropout - a different approach on preventing overfitting is to artificially reduce the complexity of the model. In this technique, during training, some of the neurons and their connections are randomly dropped, i.e. the weights are set to 0. This reduces the number of parameters and helps the algorithm to generalize better (Srivastava et al., 2014).

In this thesis, the first two regularization methods are used.

2.2 Reinforcement Learning

2.2.1 Overview

The goal of Reinforcement Learning is to learn what to do, or what actions to take, in certain situations (states), in order to maximize reward, which is a numeric signal (Sutton and Barto, 2018). Depending on the environment, this comes with complications. In some cases, the states depend on previous states or on previous actions of the agent. In others, it may be a stochastic process. Obviously, the algorithm requires a lot of trial-and-error search in order to maximize reward.

Reinforcement learning is neither supervised learning nor unsupervised learning, but rather a separate category. Supervised learning is based on learning from a training set, where the correct actions are given, and tries to generalize the information. Unsupervised learning, on the other hand, has the objective of finding hidden structure of unlabelled data with no "correct answer". Reinforcement learning has a final objective, but, unlike in supervised learning, it is not split into smaller, isolated tasks, but is only regarded as a whole. Also, rather that learning from given data, reinforcement learning algorithms learn from actions they make when interacting with the environment.

A common problem in reinforcement learning is called the exploration–exploitation dilemma. When an algorithm learned a way to earn reward, it may use only this knowledge. This, however, comes at a cost of missed opportunities of finding a more effective way to earn reward. In addition to that, the environment itself may change, making the previously optimal policy ineffective. As a result, an algorithm can sometimes deviate from the optimal policy and do explorative actions to see, how effective these are and update the policy, if necessary.

The main elements of reinforcement learning explained in more detail (Sutton and Barto, 2018):

- Policy - defines how the agent behaves at a point of time. Basically, it tells the agent, which actions to take during certain states. May be deterministic or stochastic.
- Reward - the value the reinforcement learning algorithm tries to maximize by change

its policy.

- Value function - estimates the accumulated reward earned at any given state. Gives the algorithm a method of judging its long-term success.
- Model of the environment - used for easier state predictions to minimize trial-and-error during learning.

Note that a model of the environment is not mandatory in a reinforcement learning algorithm. In this Thesis, no model of the environment is used, as no accurate model for financial time series exists.

2.2.2 Markov Decision Process

Formally, reinforcement learning follows the so-called Markov Decision Process (Sutton and Barto, 2018). The Markov Decision Process is a stochastic process with the Markov property, which means that future states depend only on the current state:

- At time step $t = 0$, the environment has an initial state $s_0 \sim p(s_0)$
- Then for $t = 0$ and later:
  - Agent selects action a_t
  - Environment outputs the reward $r_{t+1} = R(.|s_t, a_t)$
  - Environment outputs the state $s_{t+1} = P(.|s_t, a_t)$
  - Agent receives reward r_{t+1} and next state s_{t+1}

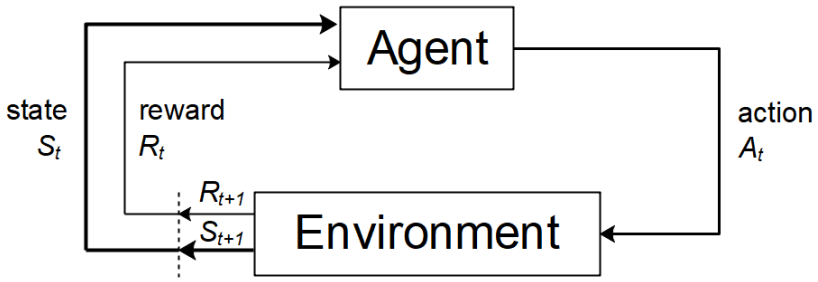


Figure 3: Illustration of a reinforcement learning process

Therefore, this interaction loop generates a sequence (trajectory) that looks like this:

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, ...$$

Then, a policy π is a function from S to A that specifies what action to take in each space. A policy is called optimal policy π^* when the accumulated discounted reward $\sum \gamma^t r_t$ is maximized. If the reward output follows a stochastic process, than the expected reward is maximized instead:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | \pi\right]$$

In the policy optimization process, two concepts have to be distinguished:

- The value function at state s shows, what is the expected cumulative reward from following the policy from state s (How good is a state?):

$$V^\pi(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

- The Q-value function at state s and action a is the expected cumulative reward from taking action a in state s and then following the policy (How good is a state-action pair?):

$$Q^\pi(s, a) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Given that, the maximum expected cumulative reward for a given state-action pair is:

$$Q^*(s, a) = \max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Q^* satisfies the following Bellman equation:

$$Q^*(s, a) = \mathbb{E}_{s' \sim \epsilon}\Big[r + \gamma \max_{a'} Q^*(s', a') | s, a\Big]$$

The equation can be explained in a way, that if the optimal state-action values for the next time step $Q^*(s', a')$ are known, the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s', a')$

Obviously, the Bellman equation can be used iteratively to eventually find the optimal policy Q^* for the whole state-action space. However, this is not feasible, since the state-action space is usually too large, so the algorithm would not scale properly.

One solution would be the application of complex algorithms, such as neural networks, to estimate $Q(s, a)$. Such a solution is called Q-learning.

2.2.3 Q-learning

Q-learning, introduced by Watkins and Dayan (1992), allows to estimate the action-value function with a function approximator (Li et al., 2017):

$$Q(s, a; \theta) \approx Q^*(s, a),$$

where θ are the parameters (weights) of the approximator.

Then, in the forward propagation step, the loss function $L_i(\theta_i)$, a simple square error, can be computed:

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}[(y_i - Q(s, a; \theta_i))^2],$$

where y_i is the "target" Q-value (since there is no true value), calculated using the Bellman equation:

$$y_i = \mathbb{E}_{s'\sim\epsilon}\Big[r + \gamma \max_{a'} Q^*(s', a'; \theta_{i-1})|s, a\Big]$$

Afterwards, the backpropagation step is realized. It updates the gradient with respect to parameters θ of the Q-function:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\epsilon}\Big[r + \gamma \max_{a'} Q^*(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \nabla_{\theta_i} Q(s, a; \theta_i)\Big]$$

Just like with the normal Bellman equation, these equations can be iterated, trying to bring the Q-value close to y, making the Q-function the optimal Q* with optimal policy π^*.

An advantage of this idea is better scalability, as, in a single learning process, all possible actions of a state are considered. This means that a neural network (or a different predictor) has multiple Q-values in the output, one for every action, and is trained on the already mentioned loss function.

Q-learning works well in scenarios where the number of actions is limited. For example, in video games, there is only a small number of different inputs to be made, and training has to be done once per frame.

In this Thesis, the number of actions per state is technically unlimited, so Q-learning will be complicated. Instead, the policy gradient algorithm will be applied.

2.2.4 Policy gradients

Sometimes, it is easier to estimate the optimal policy function directly instead of optimizing the Q-function (Li et al., 2017). To do this, the function $J(\theta)$ is defined, which can be characterized as the estimation of total reward for each policy:

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta\right]$$

Then, the objective will be to find the policy parameters, such that $J(\theta)$ is maximized.

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, J(\theta)$$

Maximization is realized by the use of gradient ascent. The gradient formula looks like this:

$$\nabla_\theta J(\theta) = \nabla_\theta \sum_\tau \pi(\tau;\theta) R(\tau) = \sum_\tau \nabla_\theta \pi(\tau;\theta) R(\tau)$$

If the policy is stochastic, it is easier to take the gradient, when transformed into a logarithm. This can be done using the likelihood ratio formula:

$$\nabla \log x = \frac{\nabla x}{x}$$

When applied to the policy function, continuing from the previous equation:

$$\nabla_\theta J(\theta) = \sum_\tau \frac{\nabla_\theta \pi(\tau;\theta)}{\pi(\tau;\theta)} \pi(\tau;\theta) R(\tau) = \sum_\tau \nabla_\theta \log \pi(\tau;\theta) \pi(\tau;\theta) R(\tau) = \mathbb{E}\Big[\nabla_\theta (\log \pi(\tau;\theta)) R(\tau)\Big],$$

where $\tau = s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, ...$ is the sequence of states, actions and rewards, mentioned earlier.

The update rule for θ is then:

$$\triangle\theta = \alpha \cdot \nabla_\theta (\log \pi(s, a, \theta)) R(\tau),$$

where α is the learning rate.

Note that, in this Thesis, the policy gradient is not stochastic, but deterministic, which makes the task of finding the optimal policy much easier. This will be explained more thoroughly in Section 3.2.4.

Since this Thesis applies policy gradients, many common gradient descent (or ascent) algorithms can be applied the same way, as in supervised learning. These will be discussed in the next section.

2.3 Optimization techniques

2.3.1 Gradient descent

In the standard gradient descent algorithm, a minimum is found for parameter θ by taking steps proportional to the negative of the gradient of the function. Then, one update step for each iteration is:

$$\theta_t = \theta_{t-1} - \alpha \nabla J(\theta),$$

where α is the learning rate and $\nabla J(\theta)$ is the gradient of the cost function.

This algorithm goes over the whole training set for each iteration, which is slow and inefficient.

A better approach is to use only a subsample, also called mini-batch, per iteration. This approach is called Stochastic gradient descent. The sample has a fixed batch size and is chosen randomly. This makes optimization faster and has an additional benefit of getting stuck less often in local minima. The disadvantage is that the absolute minimum is never found, as the Stochastic gradient descent only works with estimations of the gradient (Bottou, 1998).

$$\theta_t = \theta_{t-1} - \alpha \nabla \sum_{i=1}^{n} J(\theta, x_i),$$

By setting the batch size to 1, online learning becomes possible, since one iteration can be made for every added training sample. Both stochastic gradient descent and online learning find an application in the Thesis (see Section 3.2.5).

2.3.2 Advanced optimization techniques

Over time, more sophisticated optimization techniques were invented and implemented to speed up training. Some of the most popular ones are Momentum (Qian, 1999), RMSprop (Hinton, 2012) and Adam (Kingma and Ba, 2015).

- Momentum - in the minimizing process, Stochastic gradient descent tends to oscillate a lot. The Momentum algorithm uses, for each step, gradients from previous steps, in an attempt to cancel the oscillations and increase optimization speed. The update process looks like this:

$$v_t = \gamma v_{t-1} + \alpha \nabla J(\theta)$$

$$\theta_t = \theta_{t-1} - v_t$$

where γ is the momentum term, typically set to 0.9.

- RMSprop - an algorithm that changes the learning rate based on the gradient. It does so by dividing the learning rate by an exponentially decaying average of squared gradients. The update step is:

$$\mathbb{E}(g^2)_t = \gamma \mathbb{E}(g^2)_{t-1} + (1-\gamma) g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\mathbb{E}(g^2)_t + \epsilon}} g_t$$

  γ is recommended to be set to 0.9 here as well.

- Adam - combines the ideas of Momentum and RMSprop by manipulating both the gradient and the learning rate. First, the decaying averages of past gradients and past squared gradients are computed:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

  Basically, m_t and v_t are estimates of the first two moments respectively. However, these results are, according to the authors, biased towards zero, hence the values need to be corrected accordingly:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

  Then, the update step is:

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

The Adam optimizer is widely recognized as the best performing optimizer and is therefore used in this Thesis.

3 Model Specifics

3.1 Problem description

3.1.1 Portfolio value calculation

Given the relative price vector at time t, the portfolio value p_t without transaction costs, is calculated the following way:

$$p_t = p_{t-1}\{[(y_t - 1) \cdot w_{t-1}] + 1\},$$

where w_{t-1} is the portfolio weight vector. Each value $w_{t-1,i}$ for asset i tells the proportion of the capital invested in that particular asset. Note that this is a general formula that works for any sum of weights. That way, leveraging is made possible. For example, if the leverage is equal to 2, then the sum of the portfolio weight vector is also equal to 2.

Next, the logarithmic rate of return is calculated:

$$r_t = \log \frac{p_t}{p_{t-1}} = \log\{[(y_t - 1) \cdot w_{t-1}] + 1\}$$

Then, the final portfolio value is calculated like this:

$$p_f = p_0 \prod_{t=1}^{t_f+1} \{[(y_t - 1) \cdot \omega_{t-1}] + 1\},$$

where p_0 is initial amount of capital.

3.1.2 Transaction costs

For each time period, the portfolio needs to be rebalanced. To even maintain the weights will cost fees, they need to be adjusted for price fluctuations. Given the relative price vector, the weight vector at the end of the period w'_t is calculated like this:

$$w'_t = \frac{y_t \odot w_{t-1}}{y_t \cdot w_{t-1}}$$

To account for the transaction costs, the transaction remainder factor μ_t is introduced. Basically, it accounts for shrinking the portfolio value for each time period, such that, if p_t is the portfolio value at the beginning and p'_t at the end, the formula is:

$$p_t = \mu_t p'_t$$

Then the formulas for the log-returns and final portfolio value are respectively:

$$r_t = \log \frac{p_t}{p_{t-1}} = \log\{[(y_t - 1) \cdot w_{t-1}] + 1\}$$

$$p_f = p_0 \prod_{t=1}^{t_f+1} \{[(y_t - 1) \cdot \omega_{t-1}] + 1\},$$

The transaction remainder factor is determined by setting up an equation that includes fees paid for selling assets (reducing the weight) and buying them (increasing the weight):

$$\mu_t = 1 - (c_s + c_p - c_s c_p) \sum_{i=1}^{m} (\omega'_{t,i} - \mu_t \omega_{t,i})^+,$$

where c_p and c_s are the commission rate for purchasing and commission rate for selling, respectively. In the experiments, they both are equal to 0.25%. Note that μ_t is found both in the left-hand and the right-hand side of the equation, and solving for μ directly is impossible. Therefore an iterative algorithm is used (Ormos and Urbán, 2013):

- First, calculate an initial estimate of μ, using the formula by Moody et al. (1998)

  $$\mu^{(0)} = c \sum_{i=1}^{m} |w'_{t,i} - w_{t,i}|,$$

  where $c = c_s = c_p$

- Then iterate

  $$\mu_t^{(k)} = 1 - (c_s + c_p - c_s c_p) \sum_{i=1}^{m} (\omega'_{t,i} - \mu_t^{(k-1)} \omega_{t,i})^+,$$

  until μ converges.

3.2 The model

3.2.1 Asset pre-selection

The dataset consists of various cryptocurrency prices against Bitcoin. Using only cryptocurrencies has a major advantage in intraday algorithmic trading: cryptocurrencies are traded 24 hours a day, 7 days a week. The algorithm does not have to account for pauses or over-the-counter trading.

The data source will be Poloniex Exchange. Besides Bitcoin, of all available cryptocurrencies, 11 will be selected, which have the highest volume during the time period in the validation

set. This is necessary to ensure a high enough liquidity to fulfill two conditions, which are required to make the trading algorithm work properly and allow meaningful backtesting:

- Zero slippage - the requested trade should happen instantly and at the displayed time, when the order was placed.
- Zero market impact - the market volume should be high enough, such that the trades of the algorithm have no significant impact on the market price.

The time periods and coins, used for the experiments, are discussed in Sections 4.1 and 4.2.

3.2.2 Price tensor

The price tensor X_t is the data that is inputted in the neural network for every time period t. It has a shape of (f, n, m), where $f = 3$ is the number of features, $n = 30$ is the number of time periods and $m = 11$ is the number of assets, selected in the previous step. The three features are the close price $v_{i,t}$, the highest price $v_{i,t}^{(hi)}$ and the lowest price $v_{i,t}^{(lo)}$. However, these are normalized, since only the relative price, not the absolute price, is interesting for portfolio management:

$$V_t = \left[v_{t-n+1} \oslash v_t \middle| v_{t-n+1} \oslash v_t \middle| ... \middle| v_{t-1} \oslash v_t \middle| \mathbf{1} \right]$$

$$V_t^{hi} = \left[v_{t-n+1}^{hi} \oslash v_t \middle| v_{t-n+1}^{hi} \oslash v_t \middle| ... \middle| v_{t-1}^{hi} \oslash v_t \middle| v_t^{hi} \oslash v_t \right]$$

$$V_t^{lo} = \left[v_{t-n+1}^{lo} \oslash v_t \middle| v_{t-n+1}^{lo} \oslash v_t \middle| ... \middle| v_{t-1}^{lo} \oslash v_t \middle| v_t^{lo} \oslash v_t \right]$$

where $\oslash$ is the Hadamard division operator and $\mathbf{1} = (1, 1, ..., 1)^T$

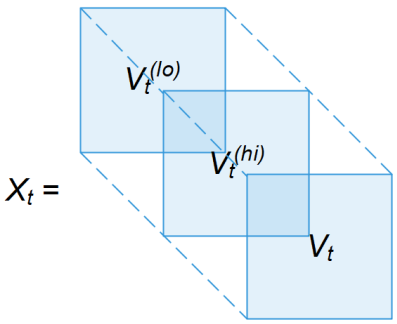


Figure 4: Illustration of the input price tensor

Note that the assets may not be available for the whole training set period. For example, Etherium only started trading in August 2015, there is no data for July 2015. In these cases, random fluctuating values are inserted, in the same way, as in Jiang et al. (2017) (i.e. random price movements with 0 decay rate).

3.2.3 Reinforcement Learning

As stated before, the main advantage of Reinforcement Learning strategies is that they do not output just buying or selling recommendations, which would be inserted into some model. Instead, the whole trading process is done automatically. In more formal terms, the Reinforcement Learning model looks like this:

- The Agent of the model is the neural network, that makes the trading decisions. In this Thesis, it will be a Convolutional neural network.
- The Action a_t of the model is the output of the neural network, i.e. the portfolio weight vector w_t, such that

$$a_t = w_t$$

- The Environment is the cryptocurrency market. It provides the agent with information, which is "high", "low" and "close" prices every 30 minutes, in the form of the price tensor X_t.
- The State s_t consists of two parts: The external state (state of the Environment), which is the price tensor X_t and the internal state (state of the Agent), which is the previously determined weight vector w_{t-1}.

$$s_t = (X_t, w_{t-1})$$

  This information is used by the Agent to determine, which action to take to earn the most reward possible.

- The Reward R is constructed as a combination of reward and punishment. The reward is equal to achieved total return r:

$$r = \frac{1}{t_f} \sum_{t=1}^{t_f+1} \log\{[(\mu_t y_t - 1) \cdot \omega_{t-1}] + 1\}$$

The punishment depends of the Maximum Drawdown (Magdon-Ismail and Atiya, 2004), the highest decline from the peak in a certain time period, which is a popular risk metric among investors:

$$D = \max_{\tau > t} \frac{p_t - p_\tau}{p_t}$$

Another important part of the punishment is the target drawdown D_{target}. Of course, it is not wise to aim for a complete elimination of risk in an investment strategy. Instead, like in Markowitz portfolio optimization, the target risk metric is determined by the investor, and the algorithm tries to not exceed the threshold D_{target}. The formula for punishment is:

$$[\max(D - D_{target}, 0)]^2$$

Note that the expression is squared to award only a mild punishment for small excesses to make sure that the algorithm converges close to the target drawdown. This also makes the reward function differentiable. Overall, the reward function is defined:

$$R = -r + [\max(D - D_{target}, 0)]^2$$

Unlike in many typical Reinforcement Learning tasks, the way to get reward is clearly defined through the reward function. In addition to that, the actions of the agent have no effect on the environment. Therefore, there is no reason to have exploration in the learning process, the algorithm can learn the optimal policy to maximize reward without worrying that there might be a function that grants even more of it. For these cases, the Deterministic Policy Gradient can be applied for learning.

3.2.4 Deterministic Policy Gradient

A policy π is a mapping from the state space to the action space, $\pi : S \to A$. Since the reward function is deterministically defined, a simple gradient ascent algorithm is enough to find the optimal policy. If a policy is defined with parameters θ, then the action $a_t = \pi_\theta(s_t)$. The performance metric is in this case, obviously, just the reward function (plus L2-regularization):

$$J_{[0,t_f]} = R(s_1, a_1, ..., s_{t_f}, a_{t_f}) + \frac{\lambda}{2m} \sum_i w_i^2$$

The policy parameters are initialized randomly. After that, they are updated along the gradient direction with a learning rate α:

$$\theta \longrightarrow \theta + \alpha \nabla_\theta J_{[0,t_f]},$$

where $\nabla_\theta J_{[0,t_f]}$ is the gradient of the performance metric.

Note that the learning process is the same, if mini-batch gradient ascent for specified sequential time ranges is used:

$$\theta \longrightarrow \theta + \alpha \nabla_\theta J_{[t_{b_1}, t_{b_2}]}$$

The mini-batch approach not only increases training efficiency, but also allows online learning. Just as described in Section 2.3.1, setting the batch size to 1 allows for additional learning, when additional samples are inserted in the data.

3.2.5 Online stochastic batch learning

As mentioned earlier, the network structure allows mini-batches with sequential inputs. Other than that, though, the mini-batches chosen for training are random. Given a fixed batch size n_b, a time point t_b is generated, such that the resulting mini-batch is $[t_b, t_b + n_b)$. Note that $[t_b, t_b + n_b)$ and $[t_b + 1, t_b + n_b + 1)$ count as two completely different mini-batches.

Due to structural changes that happen in financial markets from time to time, recent time periods are more relevant for predictions than long past ones. Therefore, the probability of the algorithm picking more recent time periods for mini-batches should be higher Holt (2004). This is realized by using a geometrically distributed probability $P_\beta(t_b)$:

$$P_\beta(t_b) = \beta(1-\beta)^{t-t_b-n_b},$$

where β is the probability-decaying rate, determining, how fast the probability decreases with each step into the past and t is the current time point.

3.2.6 Network structure

Now, the network structure to determine the optimal policy is determined. It will consist of two Convolutional neural networks, the outputs of which will be combined for the final output weight vector. The input of both these CNNs is the price tensor, which was defined earlier in Section 3.2.2.

The output of the first CNN, the "weight network", is an unleveraged weight vector, which means that the weights sum up to 1. This is achieved using a softmax function in the output layer, which looks like this (Goodfellow et al., 2016):

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1} K e^{z_k}},$$

where $j = 1, ..., K$.

The structure of the hidden layers is called by Jiang et al. (2017) an Ensemble of Identical Independent Evaluators (EIIE). Basically, this means that, when information of asset prices is passed through the hidden layers, the information for one asset never interacts with the information for other assets, rather, the interactions happen only through time. The only interaction between assets happens in the softmax layer, to ensure that the weights sum up to 1. This is supposed to improve the overall performance, since the algorithm has an easier time distinguishing between the individual assets. The other advantages of EIIE are faster learning (less weights to train), linear scaling (adding more assets in the network increases the learning time linearly) and the capability to preserve useful information for future updates, which makes online learning possible without the need to retrain the network from zero every time a time period passes.

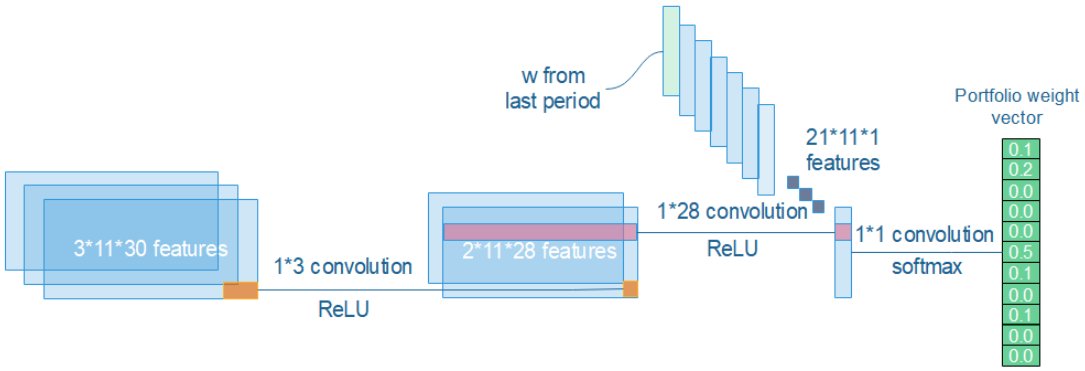


Figure 5: Structure of the weight network. The convolutions are set up to be $(1 \cdot x)$, such that different assets do not interact with each other.

The second CNN, the "leverage network", outputs a single number - the leverage coefficient, which can range between 0 and 2. To get this number, the output layer is fully connected with the sigmoid function, multiplied by 2, as activation function. The sigmoid function is defined by

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}$$

and ranges from 0 to 1. Besides the output layer, the network structure is identical to that of the "weight network".

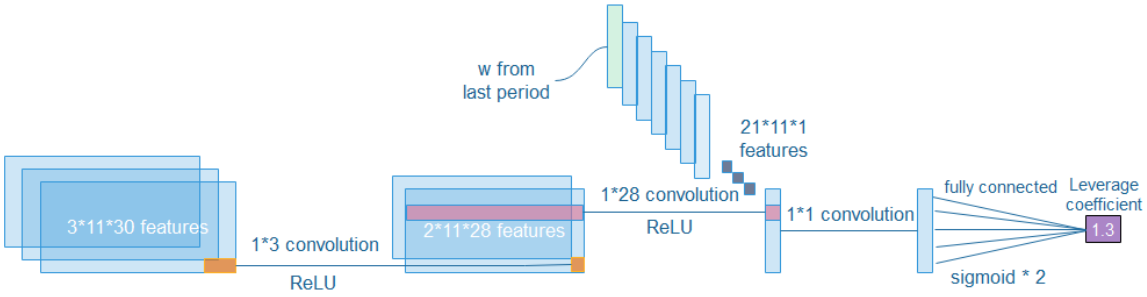


Figure 6: Structure of the leverage network. The structure is the same, besides the output layer.

The final weight vector is then computed by multiplying the output from the weight network with the output from the leverage network.

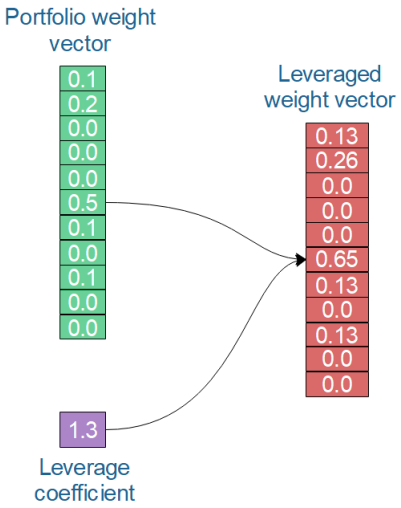


Figure 7: Calculation of final portfolio weight vector

3.2.7 Portfolio vector memory

One of the features of the neural networks in their use of the weight vector of the previous period w_{t-1} to get the current weight vector w_t. This information is drawn from the so-called portfolio vector memory, based on experience replay memory, introduced by Mnih et al. (2016). Essentially, this is a matrix, containing portfolio weight vectors for all time periods.

The portfolio vector memory needs to be trained first. It is initialized with all weights being equal to $\frac{1}{m}$ and gets updated for every time step t whenever this time step is randomly picked. After enough training steps, the portfolio vector memory converges, and can be used for predictions. The portfolio vector memory also allows the network to be trained in mini-batches.

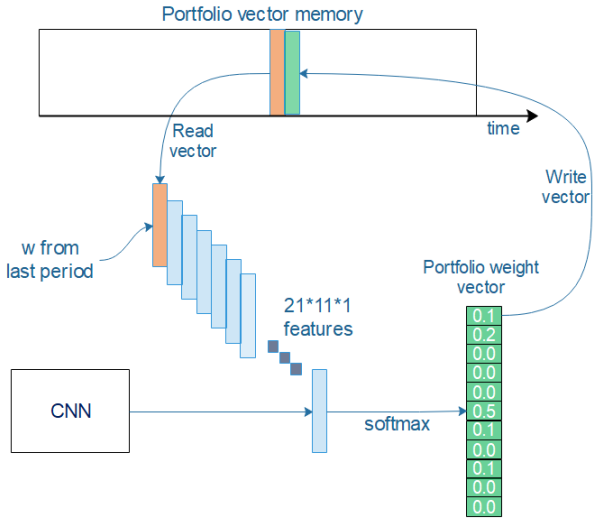


Figure 8: How Portfolio vector memory works. The weight vector from the previous time period is imported from the memory to help training the next weight vector. After finishing the calculation, the result is written in the memory for further use.

The portfolio memory can be trained further, even in the phase of online learning. This is why, every time additional data is inserted, a certain number of time periods are randomly chosen to be retrained, accounting for the new data.

In the end, this means that there are two training phases in the algorithm:

- Regular training - usage of the training set only to build up portfolio memory and network parameters
- Online learning - usage of newest available data to further refine the algorithm and adapt to new market situations

4 Results

4.1 Overview

In this thesis, there will be two main experiments to check the behaviour of the algorithm in two different economic situations.

	Experiment 1	Experiment 2
Training set	01.07.2015 - 03.04.2017	01.07.2015 - 02.08.2018
Validation set	04.04.2017 - 03.05.2017	03.08.2018 - 01.09.2018
Test set	04.05.2017 - 30.06.2017	02.09.2018 - 31.10.2018

Table 1: Data time range used for training set, validation set and test set.

The test set of the first experiment is a time where cryptocurrencies were on the rise, and some cryptocurrencies were significantly outperforming Bitcoin at that time, for example Etherium (see Figure 9). It was also a very volatile time, so it would be interesting to see, how the algorithm will regulate itself. The second experiment tests a time period that happened after severe structural changes in the cryptocurrency market. The market overall became smaller, less volatile, so, considering the high trading fees, making profits will be a much higher challenge.

Hyperparameter	Value	Description
Batch size	128	Size of mini-batch during training
Window size	30	Number of trading periods in each input price matrix
Number of assets	11	Number of preselected assets for trading
Total steps	10000-100000	Total number of steps for the first phase of training in the training set. Value depends on performance in the validation set.
Regularization coefficient	10^{-8}	The L2 regularization coefficient applied to both CNNs for all hidden layers.
Learning rate	0.0003	Step size of the Adam optimization.
Commission rate	0.25%	Rate of commission fee applied to each transaction.
Rolling steps	85	Number of online training steps for each period in the second phase of training (during the backtest).
Sample bias	$5 \cdot 10^{-5}$	Geometric distribution parameter when selecting online training sample batches.
Target Drawdown	0.1-0.5	The desired maximum drawdown the algorithm will aim for.

Table 2: Hyperparameter values used in the algorithm.

The algorithm was executed in Python. The main algorithm can be found in RL_MainComputation. The basis for the code is provided by Jiang et al. (2017).

4.2 Data

The data consists of 11 coins, selected based on the trading volume in the validation set time frame. The selected coins for experiments 1 and 2 and the trading values can be found in the table below:

	Experiment 1		Experiment 2	
	Coin	Volume (BTC)	Coin	Volume (BTC)
1	ETH - Etherium	833820.4	USDT - Tether	36412.9
2	LTC - Litecoin	669757.0	ETH - Etherium	21284.6
3	XRP - Ripple	465788.9	XRP - Ripple	13212.6
4	USDT - Tether	200241.4	STR - Stellar	8189.8
5	ETC - Etherium Classic	165836.5	XMR - Monero	6222.2
6	DASH - Dash	145825.8	ETC - Etherium Classic	5582.7
7	XMR - Monero	82908.9	DASH - Dash	4950.1
8	XEM - NEM	69048.0	LTC - Litecoin	3563.3
9	FCT - Factom	63216.5	BCH - Bitcoin Cash	3497.3
10	GNT - Golem	59158.7	ZEC - Zcash	2535.2
11	ZEC - Zcash	51503.9	DGB - DigiByte	2407.3

Table 3: Selected coins for the two experiments, ranked by trading volume. The volume is total volume in Poloniex in Bitcoin for the time period of the validation set.

The plot for the Etherium-Bitcoin exchange rate can be found below. The plots for other selected coins are in the Appendix A.1.

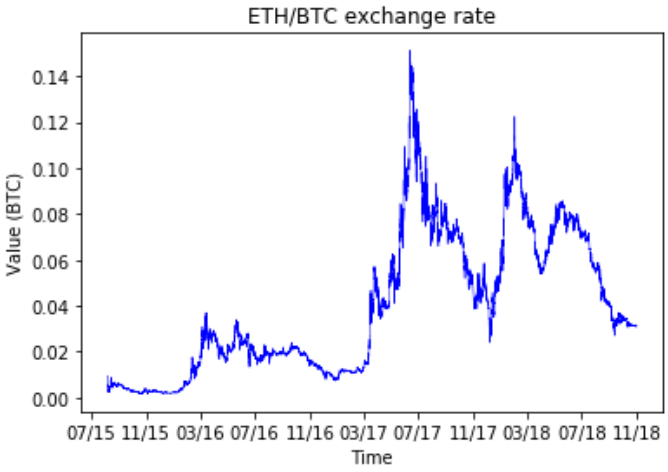


Figure 9: Etherium-Bitcoin exchange rate (from 07/2015 to 10/2018). RL_CoinFigures

Notice, how some, but not all, coins, spike around June or July of 2017, the testing period for Experiment 1. Still, an algorithm has to react fast to benefit from those spikes. Zcash had a

turbulent start, at least at Poloniex, having immense spikes at the beginning, but dropping to much lower levels since.

To make the data more meaningful, monthly returns were calculated. The descriptive statistics are presented below. The respective plots can be found in Appendix A.2.

Coin	Minimum	1st Quartile	Mean	Median	3rd Quartile	Maximum
BCH	-57.61	-15.85	2.66	-7.19	4.41	248.92
DASH	-55.82	-11.44	3.80	-2.97	12.57	216.36
DGB	-59.33	-17.01	14.90	-3.97	17.65	626.09
ETC	-43.41	-14.65	1.99	-4.59	10.45	255.05
ETH	-46.11	-14.63	7.07	-3.16	14.87	286.54
FCT	-86.44	-18.48	7.70	-5.11	12.86	566.76
GNT	-51.66	-18.84	18.53	3.92	34.74	304.91
LTC	-44.49	-10.00	0.99	-2.73	2.95	235.77
USDT	-50.28	-11.84	-2.71	-3.11	5.18	71.51
STR	-51.93	-13.90	9.63	-4.47	11.72	884.95
XEM	-48.56	-14.51	10.75	-4.31	18.11	505.96
XMR	-41.49	-11.20	7.00	-2.00	12.74	537.45
XRP	-57.68	-15.03	9.40	-5.53	6.90	705.04
ZEC	-98.96	-19.88	-2.08	-6.89	9.52	131.99

Table 4: Descriptive statistics for monthly returns (07/2015 - 10/2018, if available). All values are in percent. RL_DescriptiveStatistics

Note that, while the mean is mostly positive, the median is negative for all coins. This is the result of extreme, but rare, positive spikes.

The reference benchmark will be the CRIX, an index for cryptocurrencies (Trimborn and Härdle, 2016). However, since the reference currency in this Thesis is Bitcoin, the CRIX values also have to be converted to Bitcoin. This is easily done by applying this formula:

$$CRIX' = CRIX \cdot \frac{BTC}{USD}$$

Here is a sample plot of the resulting modified CRIX:

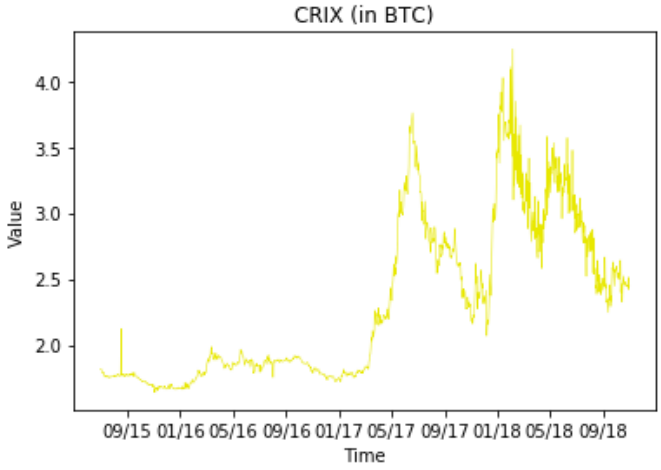


Figure 10: CRIX values, converted to Bitcoin (from 07/2015 to 10/2018)

Up to the beginning of 2017, the fluctuations were comparably low, mostly because Bitcoin was even more dominant, than it is now, as can be seen, for example, in *coinmarketcap*. Also worth mentioning is the structural break in the beginning of 2018, after which the volatility noticably increased. The peaks, where the alternative coins were most expensive, compared to Bitcoin, were around July 2017 and February 2018. This matches with the dynamics of Etherium. This makes sense, since, after Bitcoin, Etherium has the highest weight in CRIX.

4.3 Experiments

4.3.1 Experiment 1

For experiment 1, various target drawdowns D_{target} were tried. This is to compare, how both returns and drawdowns are affected, and to check, whether the limitation works as intended.

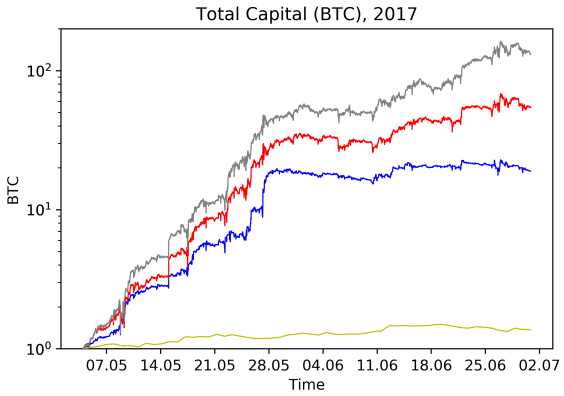


Figure 11: Total capital with D_{target} equal to 0.1, 0.4 and 0.5, compared to CRIX

RL_Experiment1Performance

	$D_{target} = 0.1$	$D_{target} = 0.4$	$D_{target} = 0.5$	CRIX
Returns (Daily)	5.30%	7.26%	8.93%	0.55%
Result (BTC)	18.98	54.39	131.15	1.37

Table 5: Returns and final capital for Experiment 1.

Naturally, the returns improved, the larger the target drawdown was. The next thing to check is, how the drawdown changes over time.

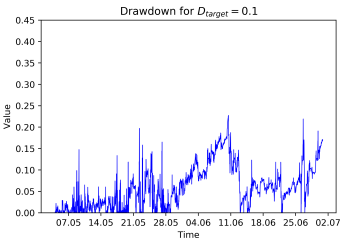


Figure 12: Drawdown for $D_{target} = 0.1$

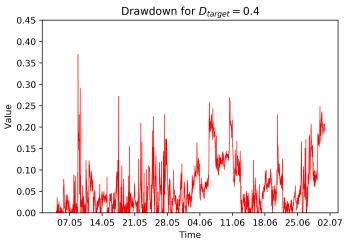


Figure 13: Drawdown for $D_{target} = 0.4$

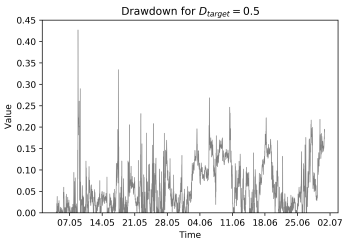


Figure 14: Drawdown for $D_{target} = 0.5$

RL_DrawdownFigures

D_{target}	0.1	0.3	0.4	0.5
MDD	0.23	0.35	0.37	0.43
Volatility	2.45	3.11	3.54	3.89

Table 6: Drawdown and annual volatility data (05/2017 - 06/2017)

It can be seen that Maximum Drawdown does decrease, the smaller D_{target}. However, at smaller values of D_{target}, the Maximum Drawdown is higher than the target drawdown. This does not mean that the algorithm failed, instead, the reason is that the algorithm trained on a different dataset, where the target drawdown is achieved, but this cannot be transferred one to one to the test set.

An interesting thing to analyse is how the leverages changes throughout the trading period. This is displayed in the following plot:

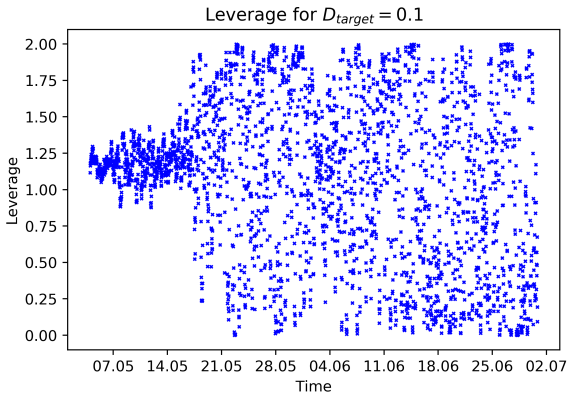


Figure 15: Leverage for $D_{target} = 0.1$. RL_Experiment1Leverage

The total leverage seems to change all the time, quickly changing between the maximum and minimum values for most of the testing period. However, it is interesting to note that in the beginning of the period, the leverage did not fluctuate too much, hovering at around 1.2 ± 0.2. This might be due to adaptations the algorithm made during learning.

Note that for $D_{target} = 0.3$ and higher, the leverage will always be very close to its maximum value of 2. At the same time, the returns and drawdowns are still different for $D_{target} = 0.3, 0.4$ and 0.5. This means the algorithm prefers to manage risk through investing in less volatile assets and through broader diversification, rather than through decreasing leverage, which

makes sense, since diversification may improve the risk-return ratio, while deleveraging can only decrease both risk and return at the same time.

The following plot visualizes the diversification for various target drawdowns:

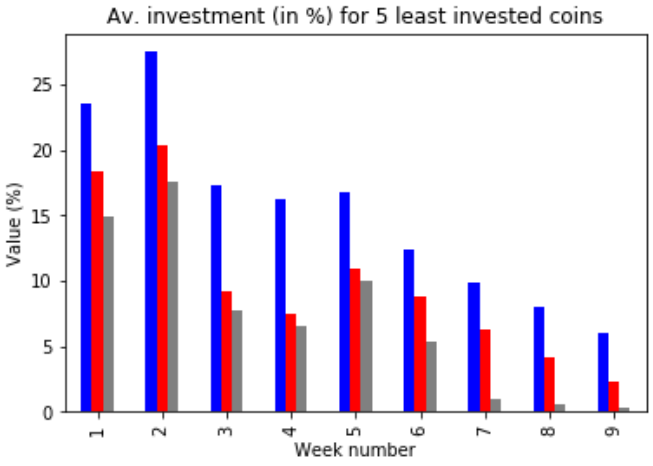


Figure 16: Average investment percentage for five least invested coins with D_{target} equal to 0.3, 0.4 and 0.5 in the test period by weeks. RL_WeightDistribution

As expected, a lower target drawdown results in higher investing even in the least popular coins. The general downwards trend is a result of continuous learning during the backtest.

4.3.2 Experiment 2

In the test period of Experiment 2, the cryptocurrency environment changed, which is easily noticed in the results.

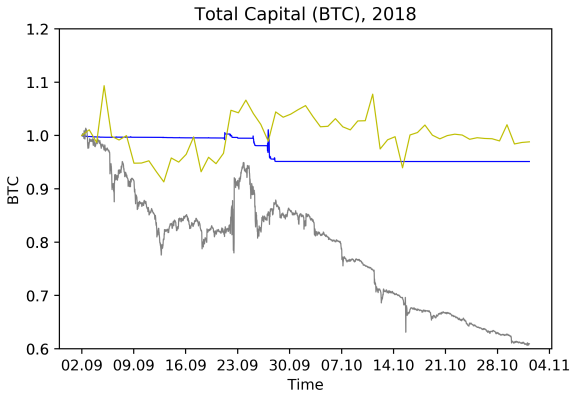


Figure 17: Total capital with $D_{target} = 0.5$ and disabled leverage, compared to CRIX. RL_Experiment2Performance

	$D_{target} = 0.5$	No leverage	CRIX
Returns (Daily)	-0.08%	-0.84%	-0.02%
Result (BTC)	0.9512	0.6094	0.9881

Table 7: Returns and final capital for Experiment 2.

Overall, it seems like the algorithm struggles to gain any profits. In fact, if there were no leverage, the algorithm would suffer losses. However, understanding those potential losses, the algorithm decided to not trade at all, with a few exceptions, therefore working as intended. This applies even to high values of D_{target}.

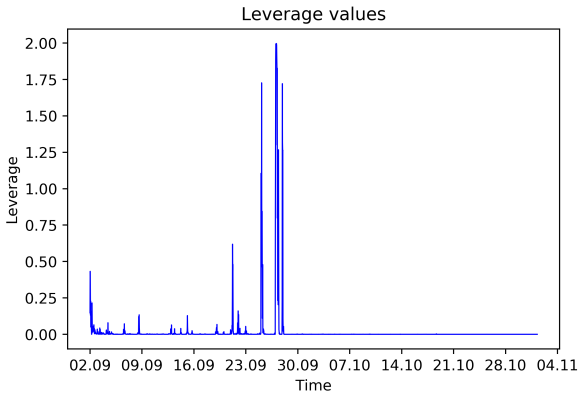


Figure 18: Leverage for $D_{target} = 0.5$. RL_Experiment2Leverage

The losses in the "no leverage" scenario are due to two reasons. First, the structure in the

cryptocurrency market changed dramatically in early 2018, making it hard to adapt to the new situation in the market. Second, the market became less exploitative, because, in real life, the trading fees decreased over time, while the algorithm still worked with a high fee of 0.25%.

5 Discussion and further research

5.1 Discussion on the methods

5.1.1 Data quality

There were no problems with the data itself. Unfortunately, the cryptocurrency market is new, and the amount of data is relatively low. As a result, some coins that were selected, did not exist in the beginning of the selected period. The missing data needed to be filled with artificial data. But even if the filled in data is a good enough imitation, there is a second problem, which is the extreme volatility that happens, when a coin first emerges. Although the algorithm takes long past events into account to a smaller extent, the final results may still be affected by this.

5.1.2 Data generalization

The cryptocurrency market is very special, as it is both predictable and unpredictable at the same time. On the one hand, the market is new and contains assets with unknown future impact on everyday life, making it very volatile. On the other hand, the market is very far from perfect, and has some inexperienced, irrational participants, which can be, in theory, easily exploited.

This Thesis primarily used cryptocurrencies because of data availability and nice properties, such as continuous trading. For closing markets, such as the stock market, the data used has to be daily. In addition to that, an assumption must be made, that closing prices match the opening prices of the next trading day. This is how it is done in the paper by Liang et al. (2018) when applying Reinforcement Learning it to the Chinese stock market. The stock market, unlike the cryptocurrency market, does not come with such drastic structural changes (although structural changes certainly still exist). This might make the adaptation to new situations easier, although overall return may be weaker.

5.1.3 Model architecture

With Reinforcement Learning, technically, any predictor could be used, not just the CNN. LSTM and RNN would also be good choices to base the predictor on. However, CNN was the best performing network in the paper of Jiang et al. (2017), so it is likely the best one for this Thesis. The goal of this Thesis, though, was to check, whether the algorithm can include risk metrics in the optimization process, not so much about finding the best structure possible. On the other hand, the "leverage" network could surely be improved upon. For example, it would be interesting to try using "leverage memory", i.e. saving leverage in addition to the portfolio weights for more efficient training.

5.2 Discussion of the results

At the first glance, the algorithm works well. It was able to learn a complex reward function, and changing target drawdown does affect the results as they should. The leverage also works as intended. It increases in favourable situations and goes to zero, if losses are expected. In the experiments, total maximum drawdown was considered when training. However, it might be more reasonable to only consider recent maximum drawdown in the future. For stock markets, it is common to only take the last 36 months into consideration, for example, when calculating the Calmar ratio (Young, 1991). For cryptocurrencies, it may make sense to make the time window even lower.

Of course, as with all trading strategies, backtests do not necessarily show, that the algorithm always performs well, and future structural breaks may significantly change the outcome.

5.3 Generative Adversarial Networks

Recently, another deep learning algorithm started to be relevant for portfolio management purposes - the Generative Adversarial Network (GAN). It was initially introduced by J. Goodfellow (2014) and was used for generating pictures with certain characteristics. Every GAN consists of a generative network G, which generates the data, and a discriminative network D, which tries to distinguish generated and real data. The networks are then trained simultaneously, such that the objective of D is to minimize its error, while the objective of G is to maximize it.

Zhou et al. (2018) uses the generative network for capturing the distribution of financial data and generating artificial stock price movements. After training with the discriminative network, the generative network can be used to generate future stock prices, therefore, essentially predicting the stock price.

6 Conclusion

In this thesis, a new Deep Reinforcement Learning model was introduced and tested on cryptocurrency data. First, the theory for CNNs and Reinforcement Learning is explained, how they work, as well as why policy gradients were chosen over Q-Learning. Then the model for financial applications was built, such that a combination of two CNNs output leveraged weights, which can be applied directly for investment purposes. The model could then be evaluated and trained using a reward function which rewards return but punishes excessive drawdown. Finally, two experiments were made with completely different test data - one for a rising market and the other for a stagnant/falling market.

For rising markets, the model restricts itself from risking too much, by both changing portfolio structure and leverage, depending on the target drawdown. Obviously, the resulting maximum drawdown still deviates from the target, begin sometimes larger and sometimes smaller. This is expected, since training data is always different from the test data, especially for cryptocurrencies. At the same time, the performance is solid and successfully beats the CRIX in terms of returns. For bearish markets, the algorithm correctly identified its incapability to have positive returns (mostly due to very high transaction costs) and stops investing, even with a generous target drawdown.

While this thesis used the model in the cryptocurrency market, it should be, with minor adjustments, applicable to other markets, such as the stock market. In fact, it could even work better, since the target drawdown should be reached more reliably in a more stable environment.

The algorithm works well in these experimental time periods, however, ideally, maximum drawdown should be measured based for a certain time window, not since inception. Some further improvements could be considered, such as allowing short-selling, or adding more features, such as volume. Of course, experiments with other metrics (Sharpe ratio etc.) as the reward function can also be made in the future.

References

Bottou, L. (1998): "Online Algorithms and Stochastic Approximations," in *Online Learning and Neural Networks*, ed. by D. Saad, Cambridge, UK: Cambridge University Press, revised, oct 2012.

Farebrother, J., M. C. Machado, and M. Bowling (2018): "Generalization and Regularization in DQN," ArXiv-preprint, index 1810.00123.

Goodfellow, I., Y. Bengio, and A. Courville (2016): *Deep Learning*, MIT Press, `http://www.deeplearningbook.org`.

Hinton, G. (2012): "Overview of mini-batch gradient descent," Unpublished lecture.

Hoerl, A. E. and R. W. Kennard (1970): "Ridge regression: Biased estimation for nonorthogonal problems," *Technometrics*, 12, 55–67.

Holt, C. (2004): "Forecasting seasonals and trends by exponentially weighted moving averages," *International journal of forecasting*, 20, 5–10.

J. Goodfellow, J. Pouget-Abadie, M. M. e. a. (2014): "Generative adversarial nets," in *28th Annual Conference on Neural Information Processing Systems 2014*, NIPS 2014, 2672–2680.

Jiang, Z., D. Xu, and J. Liang (2017): "A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem," ArXiv-preprint, index 1706.10059.

Kingma, D. P. and J. L. Ba (2015): "Adam: a Method for Stochastic Optimization," *International Conference on Learning Representations*, 1–13.

Li, F., J. Johnson, and S. Young (2017): "Deep Reinforcement Learning," Lecture notes from Stanford University.

Liang, Z., H. Chen, J. Zhu, K. Jiang, and Y. Li (2018): "Adversarial Deep Reinforcement Learning in Portfolio Management," ArXiv-preprint, index 1808.09940.

Luo, R., W. Zhang, X. Xu, and J. Wang (2017): "A Neural Stochastic Volatility Model," ArXiv-preprint, index 1712.00504.

Magdon-Ismail, M. and A. Atiya (2004): "Maximum drawdown," *Risk Magazine*, 17, 99–102.

Markowitz, H. (1952): "Portfolio Selection," *The Journal of Finance*, 7, 77–91.

Mnih, V., P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu (2016): "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning*, 1928–1937.

Moody, J., L. Wu, Y. Liao, and M. Safell (1998): "Performance functions and reinforcement learning for trading systems and portfolios," *Journal of Forecasting*, 17, 441–470.

Necchi, P. G. (2016): "Reinforcement Learning For Automated Trading," .

Ormos, M. and A. Urbán (2013): "Performance analysis of log-optimal portfolio strategies with transaction costs," *Quantitative Finance*, 13, 1587–1597.

Prechelt, L. (2012): *Early Stopping — But When?*, Berlin, Heidelberg: Springer Berlin Heidelberg, 53–67.

Qian, N. (1999): "On the momentum term in gradient descent learning algorithms," *Neural Networks : The Official Journal of the International Neural Network Society*, 12, 145–151.

Silver, D., A. Huang, and C. J. M. et al. (2016): "Mastering the game of Go with deep neural networks and tree search," *Nature*, 529, 484–503.

Sirignano, J. A. (2016): "Deep Learning for Limit Order Books," ArXiv-preprint, index 1601.01987.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014): "A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, 15, 1929–1958.

Sutton, R. S. and A. G. Barto (2018): *Reinforcement Learning: An Introduction*, Cambridge, Massachusetts: MIT Press.

Tikhonov, A. N. (1943): "Об устойчивости обратных задач [On the stability of inverse problems]," *Doklady Akademii Nauk SSSR*, 39, 195–198.

Trimborn, S. and W. K. Härdle (2016): "CRIX an Index for Blockchain Based Currencies," *SFB 649 Discussion Paper, Economic Risk*, 2016-021.

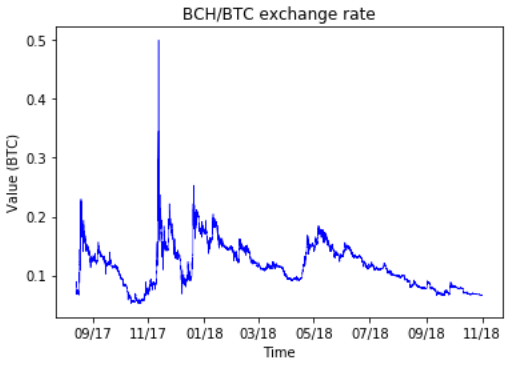
Watkins, C. J. C. H. and P. Dayan (1992): "Q-Learning," *Machine Learning*, 8, 279–292.

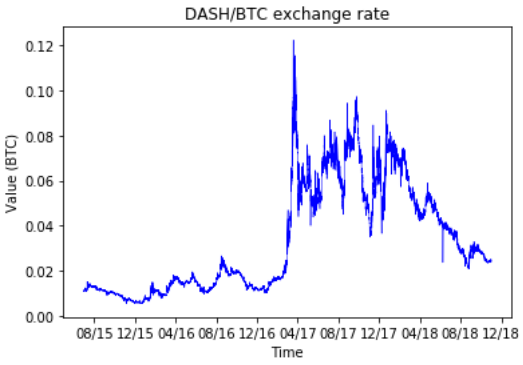
Young, T. (1991): "Calmar Ratio: A Smoother Tool," *Futures*.

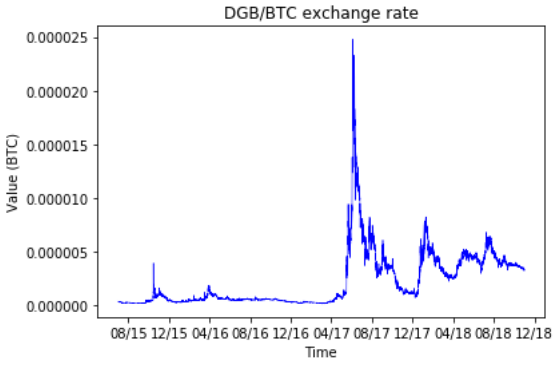
Zhou, X., Z. Pan, G. Hu, S. Tang, and C. Zhao (2018): "Stock Market Prediction on High-Frequency Data Using Generative Adversarial Nets," *Mathematical Problems in Engineering*, 2018.

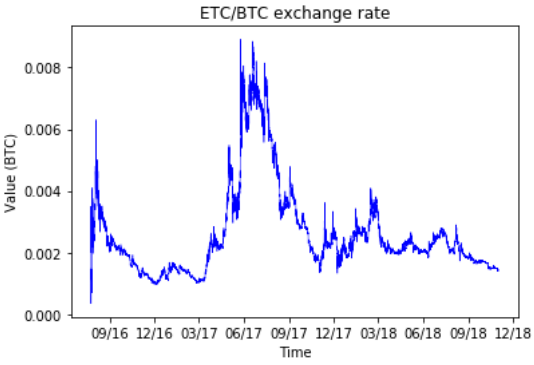
A Figures

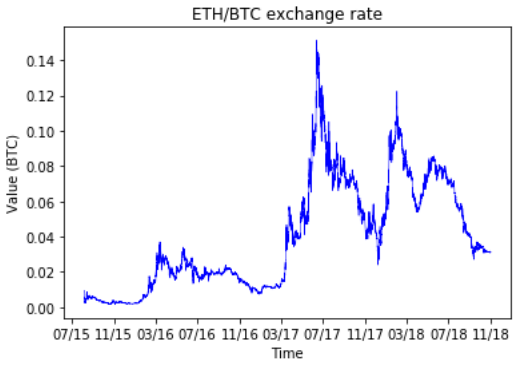
A.1 Cryptocurrencies prices

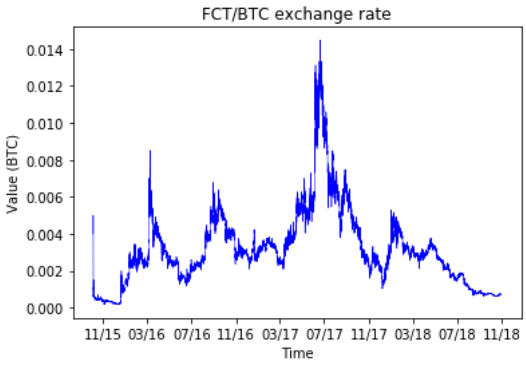


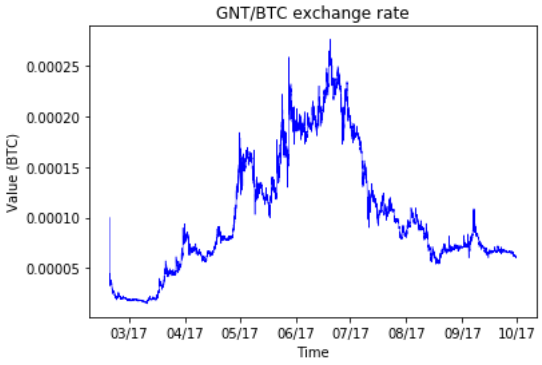


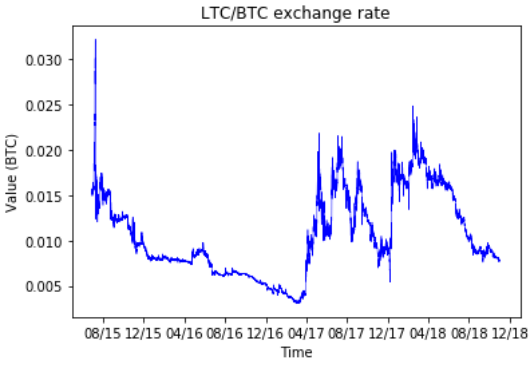




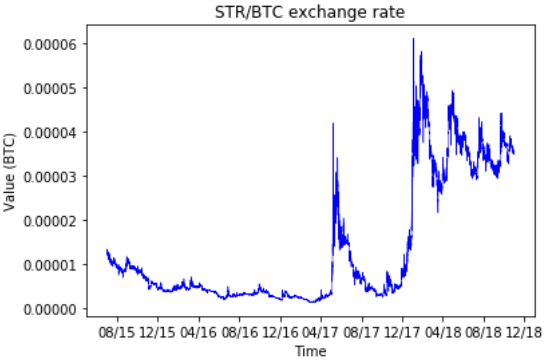


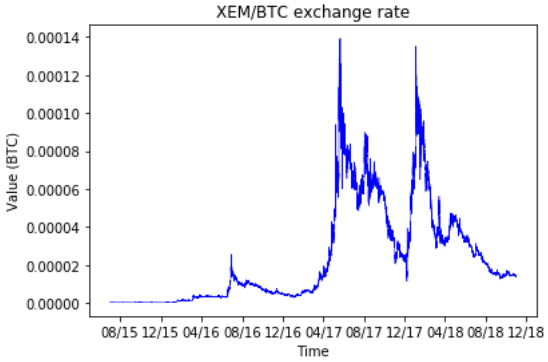


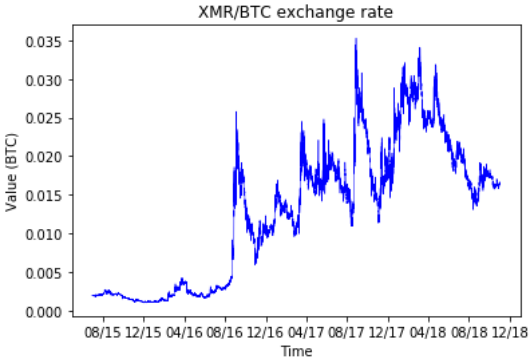

GNT/BTC exchange rate
Value (BTC)
0.00025
0.00020
0.00015
0.00010
0.00005
03/17 04/17 05/17 06/17 07/17 08/17 09/17 10/17
Time


LTC/BTC exchange rate
Value (BTC)
0.030
0.025
0.020
0.015
0.010
0.005
08/15 12/15 04/16 08/16 12/16 04/17 08/17 12/17 04/18 08/18 12/18
Time


USDT/BTC exchange rate
Value (BTC)
0.005
0.004
0.003
0.002
0.001
0.000
08/15 12/15 04/16 08/16 12/16 04/17 08/17 12/17 04/18 08/18 12/18
Time


STR/BTC exchange rate
Value (BTC)
0.00006
0.00005
0.00004
0.00003
0.00002
0.00001
0.00000
08/15 12/15 04/16 08/16 12/16 04/17 08/17 12/17 04/18 08/18 12/18
Time


XEM/BTC exchange rate
Value (BTC)
0.00014
0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
08/15 12/15 04/16 08/16 12/16 04/17 08/17 12/17 04/18 08/18 12/18
Time


XMR/BTC exchange rate
Value (BTC)
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
08/15 12/15 04/16 08/16 12/16 04/17 08/17 12/17 04/18 08/18 12/18
Time

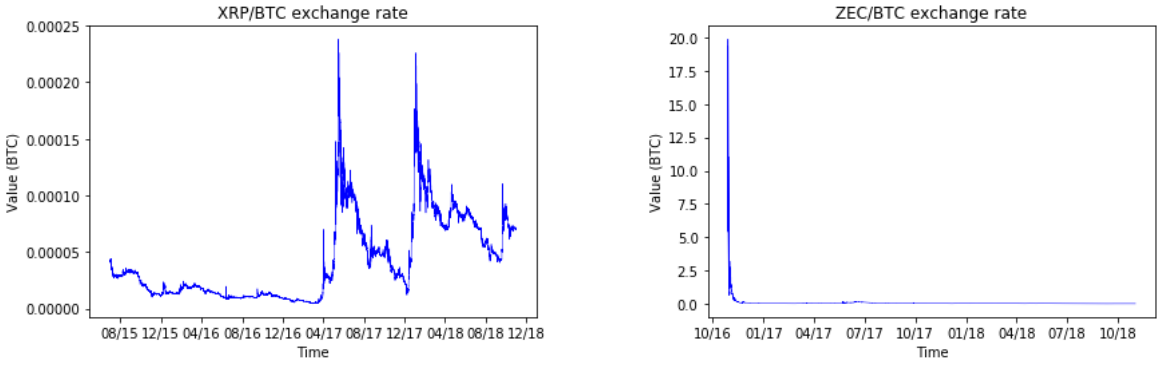
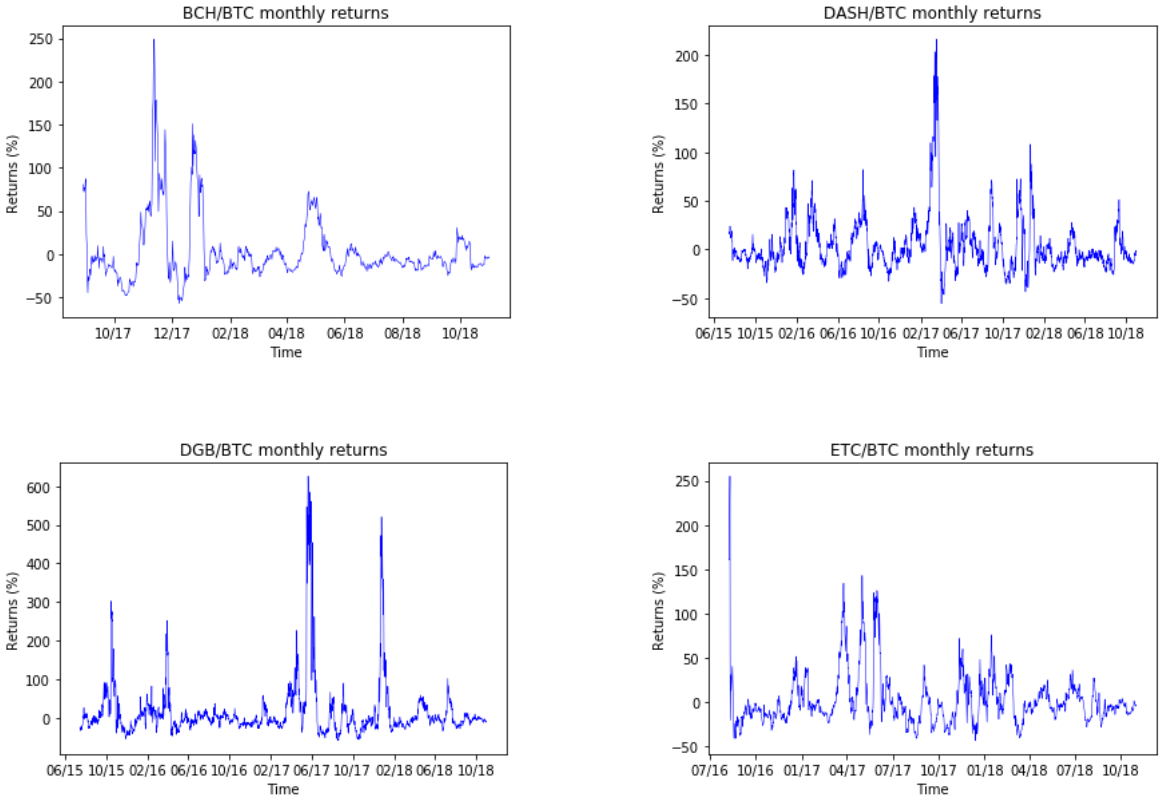
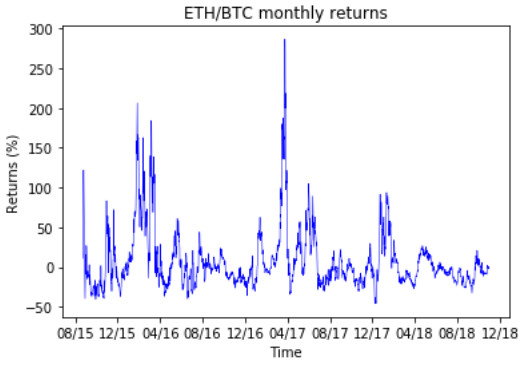
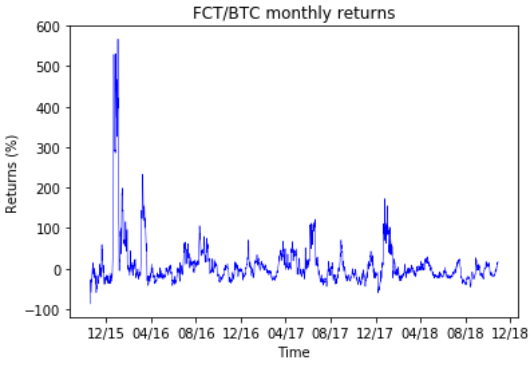


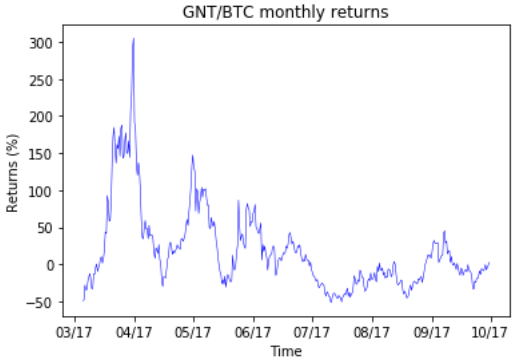
Figure 19: Exchange rates for all coins included in the experiments. The time series has a frequency of 30 minutes, contain data for the end of each period and range from July 2015 to October 2018. Some coins were not traded throughout the whole period, so the time series starts at the beginning of trading and ends when trading stopped. Note that the initial trading day (28.10.2016) for Zcash (ZEC) was extremely volatile and excluded from the plot. RL_CoinFigures

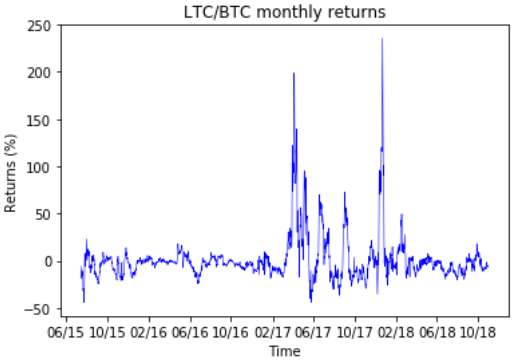
A.2 Cryptocurrencies monthly returns

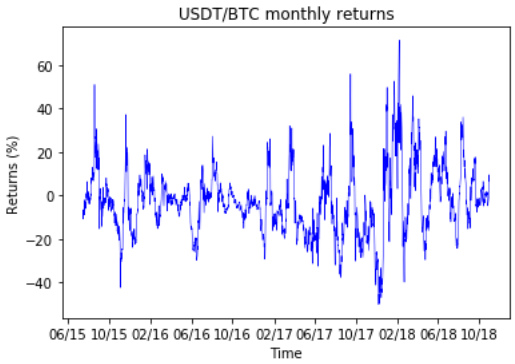


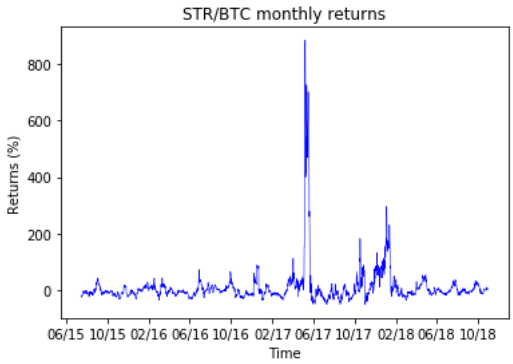

ETH/BTC monthly returns
Returns (%)
Time

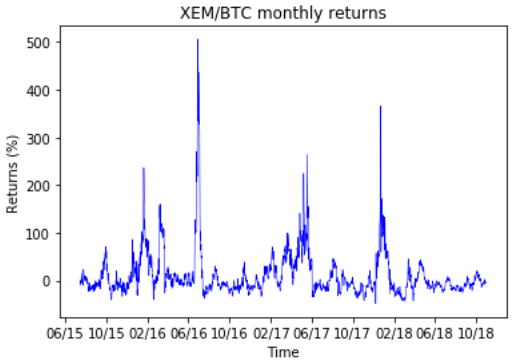

FCT/BTC monthly returns
Returns (%)
Time

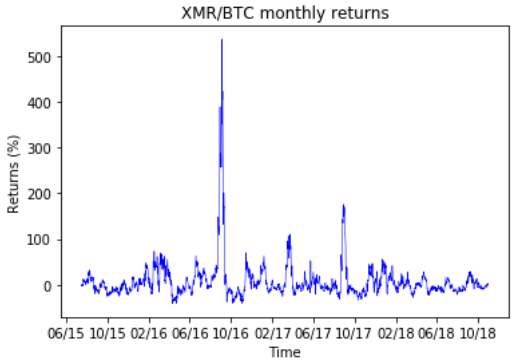

GNT/BTC monthly returns
Returns (%)
Time


LTC/BTC monthly returns
Returns (%)
Time


USDT/BTC monthly returns
Returns (%)
Time


STR/BTC monthly returns
Returns (%)
Time


XEM/BTC monthly returns
Returns (%)
Time


XMR/BTC monthly returns
Returns (%)
Time

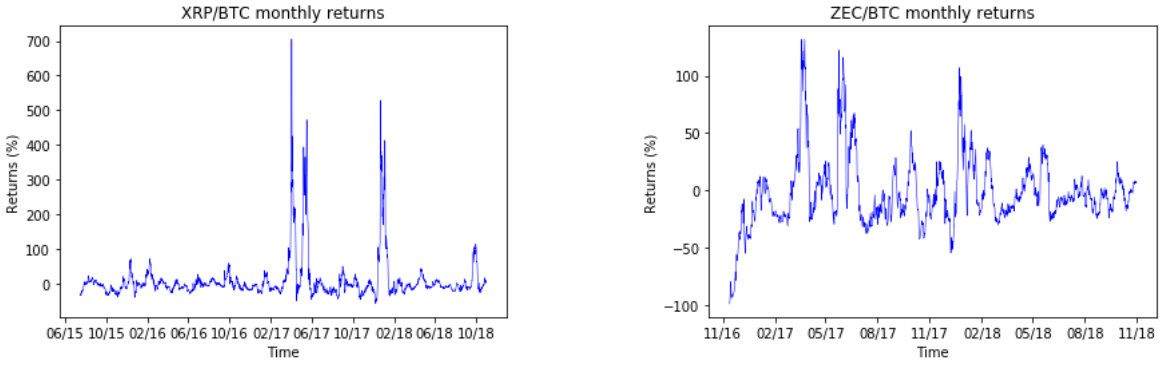


Figure 20: Return values in percent for all coins included in the experiments. The time series has a frequency of 30 minutes, contain data for the end of each period and range from July 2015 to October 2018. Some coins were not traded throughout the whole period, so the time series starts at the beginning of trading and ends when trading stopped. Note that the initial trading day (28.10.2016) for Zcash (ZEC) was extremely volatile and excluded from the plot. RL_CoinReturnsFigures